SOLID Principles

The Single Responsibility Principle (SPP)

To achieve the SRP, I make sure that every class is responsible for its own actor, for example:

In the server application:

- `EmployeeCache`: this class save items which have more usage and recently used.
- `BasicConnectionPool`: this class for database connection cache.
- `StringSocketReader`: this class reads a String value from a socket.
- `StringSocketWriter`: this class writes a String value to socket.
- `EmployeeMySqlRepository`: this class MySql database access object for Employee.
- A `Query` class for each query to the database that retrieve a single thing from the database.
- A `Request` class for each request to the server.
- `ClientRunnable`: this class that handles a single connection with a client.
- `ServerThread`: this class which contains the main functionality of the server.

In the client application:

- `MainView`: this class act as View that prints to console
- `AppMainController`: this class act as an interface between Model and View components to process all the business logic and incoming requests.
- `AppDataManager`: this class manage the whole data on the app.
- `AppDBHelper`: this class used to simplify work with the database server.
- `StringSocketReader`: this class reads a String value from a socket.
- `StringSocketWriter`: this class writes a String value to socket.
- A `Request` class for each request to the server.

The Open-Closed Principle (OCP)

This principle applies to the project since it allows you to design any kind of data to be used in the repository , where you can easily extend the functionality by writing you own entity class and implementing an API that uses the `Repository` with this new entity without any need to rewrite any code related to the `Repository` interface.

The Liskov Substitution Principle (LCP)

This principle applies to the project since it allows you to design:

- Any kind of requests by extends the functionality by writing your own `Request` class and extends `Request` class without any unexpected behaviour.
- Any kind of queries by extends the functionality by writing your own `Query` class and implementing `Query` interface without any unexpected behaviour.
- Any kind of repository by extends the functionality by writing your own `Repository` class and implementing `Repository` interface without any unexpected behaviour.
- Any kind of socket readers by extends the functionality by writing your own `SocketReader` class and implementing `SocketReader` interface without any unexpected behaviour.
- Any kind of socket writers by extends the functionality by writing your own `SocketWriter` class and implementing `SocketWriter` interface without any unexpected behaviour.
- Any kind of connection pools by extends the functionality by writing your own `ConnectionPool` class and implementing `ConnectionPool` interface without any unexpected behaviour.
- Any kind of caches by extends the functionality by writing your own `Cache` class and implementing `ICache` interface without any unexpected behaviour.

The Interface Segregation Principle (ISP)

The implementation of `EmployeeUpdateByIdQuery`, `EmployeeInsertQuery`, `EmployeeFindByIdQuery`, `EmployeeFindAllQuery` and `EmployeeDeleteByIdQuery` API depends on the `Query` interface, but they are different classes, and neither of them depends on the other, thus if we had another `Query` class we will not change the implementation of other queries.

The Dependency Inversion Principle (DIP)

In the server application:

- The `ICache` object was injected to the `ClientRunnable` by the `ServerThread`.
- The `ConnectionPool` object was injected to the `ClientRunnable` by the `ServerThread`.
- The `Repository` object was injected to the `ClientRunnable` by the `ServerThread`.
- The `Gson` object was injected to the `ClientRunnable` by the `ServerThread`.
- The `Repository` object was injected to the `Query` object by the `ClientRunnable`.

In the client application:

- The `Gson` object was injected to the `AppDBHelper` by the `Main`.
- The `DBHelper` object was injected to the `AppDataManager` by the `Main`.
- The `DataManager` object was injected to the `AppMainController` by the `Main`.
- The `MainController` object was injected to the `MainView` by the `Main`.